Safe object composition in the presence of subtyping*

Lorenzo Bettini[1] Viviana Bono[2] Silvia Likavec[2]

[1] Dipartimento di Sistemi ed Informatica, Università di Firenze, Viale Morgagni 65, 50134 Firenze, Italy, bettini@dsi.unifi.it
[2] Dipartimento di Informatica, Università di Torino, C.so Svizzera 185, 10149 Torino, Italy, {bono,likavec}@di.unito.it

Abstract. Object composition arises as a natural operation to combine objects in an object-based setting. In our incomplete objects setting it has a strong meaning, as it may combine objects with different internal states. In this paper we study how to make object composition safe in the presence of width subtyping, we propose two solutions, and discuss the alternative ones.

1 Introduction

Object composition is often advocated as an alternative to class inheritance, in that it is defined at run-time and enables dynamic object code reuse by assembling the existing components: "Ideally, you shouldn't have to create new components to achieve reuse. You should be able to get all the functionality you need just by assembling existing components through object composition." [9].

This paper is about combining safely object composition and width subtyping on objects, as their co-existence introduces run-time conflicts between methods that might have been hidden by subsumption, in situations where statically there would be no conflict. Suppose we have two objects O_1 and O_2 that we want to compose. Object O_1 has a method m_1 that calls a method m, which might be hidden by subsumption (i.e., its name does not appear in the type of the object O_1). Object O_2 has a method m_2 that calls a method m, also possibly hidden by subsumption. (Notice that it is enough if the method m is hidden in at least one of the objects.) When these two objects are composed, there is no explicit name clash at the type level, but methods m_1 and m_2 both call a method with the same name m and it is necessary: (*i*) to ensure that, after object composition, methods m_1 and m_2 continue calling the method m they were calling originally, before the composition; (*ii*) to guarantee that if one of the m's is not hidden, we expose the reference to the right one in the resulting object's "public interface". Note that if both m's were hidden by subsumption, none of them would be available to the external users anymore, and if none were hidden there would be a *true* conflict, ruled out statically.

This situation is an instance of the "width subtyping versus method addition" problem (well known in the object-based setting, see for instance [8]). This kind of name clash (named *dynamic name clash*, as opposed to the above mentioned true conflict) should not be considered an error, but we must make sure that we solve all ambiguities, in such a way that accidental overrides do not occur.

* This work has been partially supported by MIUR project EOS.

We tackle this problem in Bono et al.'s calculus of classes and objects [4] setting, enriched with *abstract* classes (i.e., classes that can declare some methods without providing their implementation) and *incomplete* objects. In our calculus, abstract classes can be seen as *incomplete classes*, and their instances are *incomplete objects* that can be completed in an object-based fashion (by providing the bodies for the abstract methods). On the one hand, since our primary goal is to model object composition in the presence of subtyping (on complete objects only), we decided not to model any form of class-based inheritance, this being an orthogonal issue. On the other hand, we decided to work on a hybrid calculus, instead of on a pure object-based calculus, because: (*i*) abstract classes give rise to a natural notion of incomplete objects; (*ii*) this is a study we would like to incorporate in the work presented in [1, 2], where we introduced a hybrid mixin-based calculus, with the aim to integrate flexible inheritance mechanisms both at the level of classes and of objects. The present calculus is simpler than the mixin-based one, but the conflict composition-subtyping we introduce and solve is the same as in the mixin-based one.

Incomplete objects can be completed in an object-based fashion via *method addition* and/or *object composition* (that composes an incomplete object with a complete one), thus providing a form of object-based inheritance. Our form of method addition does not introduce any problems with respect to subtyping, as we can add one by one only those methods that are required explicitly by the incomplete object, that is, we have total type information about the methods to be added (coming directly from the corresponding well-typed classes) *before* the actual addition takes place (see Sections 4 and 5). The conflict arises, instead, with object composition where the complete object may have more methods than the ones required by the incomplete object, and these methods may clash with some of the methods defined in the incomplete object. Notice that this problem is exactly the same as the one introduced by the general object composition example described above. Our approach to solving this problem is based on the idea of preserving the object generator together within each object. In order to avoid undesired interactions between methods while allowing the expected rebinding, every object carries the list of its methods and the list of the methods that it is still expecting.

One of the possible approaches to solving the problem seemed to be exploiting the *dictionaries* of Riecke and Stone [11]. Unfortunately, their mapping "internal label-external label" does not solve completely the ambiguities introduced by object composition in the presence of subtyping described above. In particular, there is still an ambiguity when only one of the *m*'s is hidden by subsumption. It has to be said, however, that the original dictionaries setting is stateless, therefore method composition can be simulated by successive method additions, and dictionaries would be sufficient to model object composition. In our setting, instead, all objects (complete and incomplete) have a state (i.e., an initialized field), and object composition cannot be linearized via any form of repeated method additions. On a side note which will be useful later, we would like to recall that the calculus of [11] is "late-binding", i.e., the host object is substituted to self (in order to solve the self autoreferences) at method-invocation time, whereas our calculus is "early-binding", i.e., the host object is bound to self at object-creation time.

To the best of our knowledge, it is not possible to remove all the ambiguities without either carrying along the additional information on the methods hidden by subsumption,

$$
\begin{array}{l}
e ::= const \mid x \mid \lambda x.e \mid e_1\ e_2 \mid fix \\
\quad \mid \mathsf{ref} \mid\ ! \mid := \mid \{x_i = e_i\}^{i\in I} \mid e.x \mid \mathsf{H}\ h.e \\
\quad \mid \mathsf{class} \\
\qquad \mathsf{method}\ m_j = v_{m_j};\ ^{(j\in \mathsf{M})} \\
\qquad \mathsf{abstract}\ m_i;\ ^{(i\in \mathsf{A})} \\
\qquad \mathsf{constructor}\ v_c \\
\qquad \mathsf{end} \\
\quad \mid \mathsf{classval}\langle Gen_c, \mathsf{M}, \mathsf{A}\rangle \mid \mathsf{new}\ e \\
\quad \mid \mathsf{obj}\langle v_g, \mathsf{M}, \mathsf{A}\rangle \\
\quad \mid \mathsf{obj}\langle v_g, \mathsf{M}, \{m_i = v_{m_i}\}^{i\in \mathsf{M}}\rangle \\
\quad \mid e_1 \longleftarrow m_i = e_2 \mid e_1 \longleftarrow e_2
\end{array}
\qquad
\begin{array}{l}
v ::= const \mid x \mid \lambda x.e \mid fix \mid \mathsf{ref} \mid\ ! \\
\quad \mid := \mid := v \mid \{x_i = v_i\}^{i\in I} \\
\quad \mid \mathsf{classval}\langle Gen_c, \mathsf{M}, \mathsf{A}\rangle \\
\quad \mid \mathsf{obj}\langle v_g, \mathsf{M}, \mathsf{A}\rangle \\
\quad \mid \mathsf{obj}\langle v_g, \mathsf{M}, \{m_i = v_{m_i}\}^{i\in \mathsf{M}}\rangle
\end{array}
$$

Table 1. Syntax of the core calculus: expressions and values.

or restricting the width subtyping. We discarded immediately the solution of re-labelling method names at object composition time, as this is untidy from a semantical point of view and impractical from an implementation one. (By re-labelling we mean the actual physical renaming of method names, and therefore all method invocations within method bodies.)

In this version of the calculus, we decided to allow width subtyping only on *complete objects*, and we present two solutions for the object composition problem. The first one is an "early-binding" version of the dictionaries approach, where the notion of "privacy-via-subsumption" of [11] is completely implemented (see Sections 4 and 5). The second one solves the conflict "method composition versus width subtyping" by relaxing the above mentioned notion (see Section 6). We argue that the first solution is more elegant formally, while the second solution is less restrictive from the point of view of the typing, and it might give better performances if implemented. In Section 7 we hint to other possible solutions of the conflict "method composition versus width subtyping".

2 Syntax

Starting from the imperative calculus of classes and objects of [4], we add the constructs to work with incomplete objects. The lambda-calculus related forms in Table 1 are standard. We describe below the other forms:

- ref, !, := are operators[3] for defining a reference to a value, for dereferencing a reference and for assigning a new value to a reference, respectively.
- $\{x_i = e_i\}^{i\in I}$ is a record and $e.x$ is the record selection operation.
- h is a set of pairs $h ::= \{\langle x, v\rangle^*\}$, where x is a variable and v is a value (first components of the pairs are all distinct). The set of pairs h is the *store*, or *heap*, found in the expression form $\mathsf{H}h.e$, where it is used for evaluating imperative side effects. In the expression $\mathsf{H}\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e$, H binds variables $x_1, \ldots x_n$ in $v_1, \ldots, v_n$ and in e.

[3] Introducing ref, !, := as operators rather than standard forms such as ref$e$, !e, $:=e_1e_2$, simplifies the definition of evaluation contexts and proofs of properties. As noted in [12], this is just a syntactic convenience, as it is the curried version of :=.

– class
  method $m_j = v_{m_j}$; $^{(j \in \mathsf{M})}$
  abstract m_i; $^{(i \in \mathsf{A})}$
  constructor v_c
  end
  is a class written directly by the programmer. It contains two sorts of method declarations: methods m_j are the methods implemented in the class and methods m_i are the names of abstract methods introduced by the class. Each method body v_{m_j} is a function of a private field, *field*, and of *self*, which will be bound to the newly created object at instantiation time. Notice that the field does not appear explicitly in the syntax, as we model it as a lambda-abstracted variable in method bodies. For the sake of simplicity, we consider only one (private) field for each class, but this is not a restriction, as the field could be a tuple. Also, the calculus does not enforce the field to be available to all the methods of the class, but this is easily obtained by declaring it to be of type ref. If so, the field behaves like a proper instance variable (it is non-accessible, not only non-visible). To understand fully how, we refer the reader to [12]. The constructor value v_c is a function of one argument that returns the initialization value f for the private field (see Section 4 for its usage).
– classval$\langle Gen_c, \mathsf{M}, \mathsf{A} \rangle$ is a *class value*, the result of evaluating a class expression. The function Gen_c is the *generator* used to generate its instances, the set M contains the indices of the methods defined in the class, and the set A contains the indices of the class' abstract methods. In our calculus method names are of the shape m_i, where i ranges over an index set. They are univocally identified by the index, i.e., $m_i = m_j$ if and only if $i = j$. Therefore, method names are identified with their indices.
– new e creates a function that returns a new incomplete object.
– obj$\langle v_g, \mathsf{M}, \mathsf{A} \rangle$ is an incomplete object, where v_g is a generator function, M contains the indices of the methods defined in the class, and the set A contains the indices of the class's abstract methods. If the set A is empty the incomplete object becomes a complete object.
– obj$\langle v_g, \mathsf{M}, \{m_i = v_{m_i}\}^{i \in \mathsf{M}} \rangle$ is a fully-fledged object that is obtained by completing an incomplete object or by reducing an object obtained via instantiation of a class that does not contain abstract methods. Its first component is a generator function (kept also for complete objects, since they can be used to complete the incomplete ones), the second component M contains the indices of the methods of the object, and the third component is the record of invocable methods.
– $e_1 \longleftarrow m_i = e_2$ is the method addition operation: it adds the definition of method m_i with body e_2 to the (incomplete) object to which e_1 evaluates. It associates to the left.
– $e_1 \longleftarrow e_2$ is the object composition operation: it composes the (incomplete) object to which e_1 evaluates with the complete object to which e_2 evaluates. It associates to the right.

Class values and object forms are not intended to be written directly, but are used to define the semantics of programs.

3 Examples

In this section, we provide some examples to show how incomplete objects, and object completion via method addition and object composition, can be used to design complex systems.

For readability, we will use here a slightly simplified syntax with respect to the calculus presented in Section 2: (*i*) the method parameters are listed in between "()"; (*ii*) $e_1; e_2$ is interpreted as $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2,\ x \notin FV(e_2)$, coherently with a call-by-value semantics; (*iii*) references are not made explicit, thus $\mathsf{let}\ x = e\ \mathsf{in}\ x.m()$ should be intended as $\mathsf{let}\ x = \mathsf{ref} e\ \mathsf{in}\ (!x).m()$; (*iv*) method bodies are only sketched. Finally, $x \longleftarrow e$ should be intended as $x := (x \longleftarrow e)$.

In the first example, we present a scenario where it is useful to add some functionalities to existing objects. Let us consider the development of an application that uses widgets such as graphical buttons, menus, and keyboard shortcuts. These widgets are usually associated to an event listener (e.g., a callback function), that is invoked when the user sends an event to that specific widget (e.g., one clicks the button with the mouse or chooses a menu item).

The design pattern *command* [9] is useful for implementing these scenarios, since it allows parameterization of widgets over the event handlers, and the same event handler can be reused for similar widgets (e.g., the handler for the event "save file" can be associated with a button, a menu item, or a keyboard shortcut). However, in such a context, it is convenient to simply add a function without creating a new class just for this aim. Indeed, the above mentioned pattern seems to provide a solution in pure class-based languages that normally do not supply the object method addition operation.

Within our approach, this problem can be solved with the language constructs for method addition and completion (in order to provide further functionalities needed by the prototype). For instance, we could implement the solution as in Table 2. The incomplete object `button` expects a method `onClick` that is internally called when the user clicks on the button (e.g., by the window where it is inserted, in our example the dialog `mydialog`). The incomplete object is then completed with the event listener `ClickHandler` (by using method addition). This listener is a function that has the parameter `doc` already bound to the application main document. At this point the object is completed and we can call methods on it. Notice that the added method can rely on methods of the host object (e.g., `setEnabled`). The same listener can be installed (by using method addition again) to other incomplete objects, e.g., the menu item `"Save"` and the keyboard shortcut for saving functionalities. Moreover, since we are able to act directly on instances here, our proposal enables customization of objects at run-time.

Another way to implement the same functionalities is via object composition. For instance, if saving the document requires further and complex operations, instead of including all of these in a method, it can be more convenient to include them in an object (with other methods than the one requested by the incomplete object). In particular, the incomplete object only requires the method `onClick`: the object used for completion can have more methods (hidden by subsumption). Moreover, the additional methods will be hidden in order to avoid name clashes.

For instance, we can define the class:

```
let Button =                  let MenuItem =              let ShortCut =
 class                         class                       class
  method display = ...          method show = ...           method setEnabled = ...
  method setEnabled = ...       method setEnabled = ...     abstract onClick;
  abstract onClick;             abstract onClick;           ...
  ...                           ...                        end in
 end in                        end in
     let ClickHandler =
      (λ doc. λ self. ... doc.save() ... self.setEnabled(false)) mydoc
       in
        let button = new Button("Save") in
        let item = new MenuItem("Save") in
        let short = new ShortCut("Ctrl+S") in
          button ←+ (OnClick = ClickHandler);
          button.display();
          button.setEnabled(true);
          mydialog.addButton(button); // now it is complete
          item ←+ (OnClick = ClickHandler);
          item.setEnabled(true);
          mymenu.addItem(item);
          short ←+ (OnClick = ClickHandler);
          short.setEnabled(true);
          system.addShortCut(short);
```

Table 2. Widgets and event handler.

```
let SaveDocument =
 class
  method onClick = λdoc.λself....
  method format = λdoc.λself....
  method save = λdoc.λself....
  method compress = λdoc.λself....
  method display = λdoc.λself....
  constructor λdoc.ref doc
 end in
```

If we instantiate this class we obtain a complete object (since there are no abstract methods), that can be used to complete the incomplete objects in Table 2. In particular, the method `display` in the complete object type will be hidden by subsumption, therefore it will not interfere with the method `display` of the class `Button` (indeed, they perform different operations). Notice that the constructor of `SaveDocument` returns a reference to the passed document instance; this is the private field that all the methods in the class can use.

4 Operational semantics

Our approach is the one of giving the calculus a semantics as close as possible to an implementation. This has the advantage of minimizing the gap between the formal semantics and the actual implementation, thus reducing the risk of introducing errors

$$
\begin{array}{rlr@{\qquad}rlr}
const\ v & \to \delta(const, v) & (\delta) & \mathsf{ref} v & \to \mathsf{H}\langle x, v\rangle.x & (\mathrm{ref}) \\
\text{if } \delta(const, v) & \text{ is defined} & & \mathsf{H}\langle x, v\rangle h.R[!x] & \to \mathsf{H}\langle x, v\rangle h.R[v] & (\mathrm{deref}) \\
(\lambda x.e)\ v & \to [v/x]\ e & (\beta_v) & \mathsf{H}\langle x, v\rangle h.R[:=xv'] & \to \mathsf{H}\langle x, v'\rangle h.R[v'] & (\mathrm{assign}) \\
fix\ (\lambda x.e) & \to [fix(\lambda x.e)/x]e & (\mathrm{fix}) & R[\mathsf{H}\ h.e] & \to \mathsf{H}\ h.R[e],\ \ R \neq [\,] & (\mathrm{lift}) \\
\{\ldots, x = v, \ldots\}.x & \to v & (\mathrm{select}) & \mathsf{H}\ h.\mathsf{H}\ h'.e & \to \mathsf{H}\ h\ h'.e & (\mathrm{merge})
\end{array}
$$

Table 3. Reduction rules for standard expressions and heap expressions

$$
\begin{array}{l}
R ::= [\,] \mid R\ e \mid v\ R \mid R.x \mid \mathsf{new}\ R \mid R \diamond e \mid v \diamond R \mid R \longleftarrow m = e \mid R \longleftarrow e \mid v \longleftarrow m = R \mid v \longleftarrow R \\
\quad \mid \{m_1 = v_{m_1}, \ldots, m_{i-1} = v_{m_{i-1}}, m_i = R, m_{i+1} = e_{m_{i+1}}, \ldots, m_n = e_{m_n}\}^{1 \le i \le n}
\end{array}
$$

Table 4. Reduction contexts

caused by implementation issues. In fact, with this semantics, a direct implementation of our calculus in a functional programming language is quite straightforward (we are working on the implementation in OCaml).

The formal operational semantics is a set of rewriting rules including some standard rules for a lambda calculus with store, and some rules that evaluate the object-oriented related forms to records and functions, according to the object-as-record approach and Cook's class-as-generator-of-object principle. This operational semantics can be seen as something close to a denotational description for objects and classes, and this "identification" of implementation and semantical denotation is, in our opinion, a good by-product of our approach. The semantics is also intuitive since it is based on functions and records.

The operational semantics extends the one of the core calculus of classes and objects [4], therefore exploits the *Reference ML* of Wright and Felleisen [12] treatment of side-effects. To abstract from a precise set of constants, we only assume the existence of a partial function $\delta : Const \times ClosedVal \rightharpoonup ClosedVal$ that interprets the application of functional constants to closed values and yields closed values.

In Table 3, R's are *reduction contexts* [5, 6, 10] and their definition can be found in Table 4. Reduction contexts are necessary to provide a minimal relative linear order among the creation, dereferencing and updating of heap locations, since side effects need to be evaluated in a deterministic order. We assume the reader is familiar with the treatment of imperative side-effects via reduction contexts and we refer to [3, 12] for a description of the related rules.

The meaning of the class related rules in Table 5 is as follows. The rule (*class*) turns a class *expression* into a class *value* (notice that objects are created by instantiating class values). Given the parameter x for the constructor v_c of the class expression, the class generator returns a (partial) object generator that passes to the private field of the method bodies v_{m_j} the value f (returned by the constructor v_c). Recall that method bodies take parameters *field* and *self*. The record returned by the object generator has "dummy" method bodies for abstract methods, in such a way the generator is thus a function from *self* to *self*. Also, for all the methods in all generator functions, the method bodies are wrapped inside $\lambda y.\cdots y$ to delay evaluation in our call-by-value calculus. The above generator is called "partial" since it returns an object that contains abstract methods that cannot be invoked (present as "dummy" methods). The actual im-

$$\begin{array}{l}\textsf{class}\\ \quad \textsf{method}\ m_j = v_{m_j};\ {}^{(j\in\mathsf{M})}\\ \quad \textsf{abstract}\ m_i;\ {}^{(i\in\mathsf{A})}\\ \quad \textsf{constructor}\ v_c\\ \textsf{end}\\ \textsf{where}\\ Gen_c \triangleq \lambda x.\textsf{let f} = v_c(x)\ \textsf{in}\ \lambda self.\left\{\begin{array}{ll} m_j = \lambda y.v_{m_j}\ \textsf{f}\ self\ y & {}^{(j\in\mathsf{M})}\\ m_i = \lambda y.\ self.m_i\ y & {}^{(i\in\mathsf{A})}\end{array}\right\}\end{array} \qquad \textsf{class}\langle Gen_c,\mathsf{M},\mathsf{A}\rangle \rightarrow \quad (\textit{class})$$

$$\textsf{new class}\langle Gen_c,\mathsf{M},\mathsf{A}\rangle \rightarrow \lambda w.\textsf{obj}\langle(Gen_c\ w),\mathsf{M},\mathsf{A}\rangle \qquad (\textit{new class})$$

Table 5. Reduction rules for class related forms

$$\textsf{obj}\langle v_g,\mathsf{M},\{\ldots,m_i = v_{m_i},\ldots\}\rangle.m_i \rightarrow v_{m_i} \qquad (\textit{obj sel})$$

$$\begin{array}{l}\textsf{obj}\langle v_g,\mathsf{M},\mathsf{A}\rangle \longleftarrow\!\!+\ (m_l = v_{m_l}) \rightarrow\\ \quad \textsf{let incgen} = \lambda self.\left\{\begin{array}{ll} m_j = \lambda y.\ (v_g\ self).m_j\ y & {}^{(j\in\mathsf{M})}\\ m_l = \lambda y.\ v_{m_l}\ self\ y & \\ m_i = \lambda y.\ self.m_i\ y & {}^{(i\in\mathsf{A}-\{l\})}\end{array}\right\}\ \textsf{in}\\ \quad\quad \textsf{obj}\langle\textsf{incgen},\mathsf{M}\cup\{l\},\mathsf{A}-\{l\}\rangle \quad \textsf{where}\ l\in\mathsf{A}\end{array} \qquad (\textit{meth add})$$

$$\begin{array}{l}\textsf{obj}\langle v_g,\mathsf{M},\mathsf{A}\rangle \longleftarrow\!\!+\ \textsf{obj}\langle v'_g,\mathsf{P},\{m_i = v_{m_i}\}^{i\in\mathsf{P}}\rangle \rightarrow\\ \quad \textsf{let incgen} = \textsf{let gen}_1 = \lambda s_1.\lambda s_2.\left\{\begin{array}{ll} m_l = \lambda y.\ (v'_g\ s_2).m_l\ y & {}^{(l\in\mathsf{P}-\mathsf{A})}\\ m_r = \lambda y.\ s_1.m_r\ y & {}^{(r\in\mathsf{P}\cap\mathsf{A})}\end{array}\right\}\ \textsf{in}\\ \quad\quad \lambda self.\left\{\begin{array}{ll} m_j = \lambda y.\ (v_g\ self).m_j\ y & {}^{(j\in\mathsf{M})}\\ m_i = \lambda y.\ (v'_g\ fix(\textsf{gen}_1\ self)).m_i\ y & {}^{(i\in\mathsf{A})}\end{array}\right\}\ \textsf{in}\\ \quad\quad\quad \textsf{obj}\langle\textsf{incgen},\mathsf{M}\cup\mathsf{A},fix(\textsf{incgen})\rangle\end{array} \qquad (\textit{obj comp})$$

$$\textsf{obj}\langle v_g,\mathsf{M},\emptyset\rangle \rightarrow \textsf{obj}\langle v_g,\mathsf{M},fix(v_g)\rangle \qquad (\textit{completed})$$

Table 6. Reduction rules for object manipulation

plementation of these methods can be provided by (*meth add*), and/or (*obj comp*) given in Table 6.

The rule (*new class*) creates incomplete objects from class values. First, it applies the class generator Gen_c to an argument w, thus initializing the private field in the methods defined in the class and providing access to the object generator, that is a function from *self* to a record of methods. The application of the fixpoint operator to the object generator will create a recursive record of invokable methods (when the object is complete, see rule (*completed*) in Table 6).

The rules in Table 6 are the basic rules for manipulating objects. The rule (*obj sel*) performs method invocation on a complete object. The rules used on incomplete objects enable completing them with the method definitions they need. The rule (*meth add*) adds to an incomplete object a method m_l not yet present in the object (but required). The newly created generator function incgen (incremental generator) maps *self* to a record of methods, where concrete method definitions are taken from the object gen-

erator v_g, the abstract methods (excluding m_l) remain "dummy", and the method m_l is added. The incgen function is part of the reduct because it must be carried along in the evaluation process, in order to enable future method additions and/or object compositions. The only requirement for m_l is that the body v_{m_l} must be a function of *self*.

The rule (*obj comp*) combines two objects in such a way that the object o_2 (which must be already complete) completes the incomplete object o_1 and makes it fully functional. After completion, it will be possible to invoke all the methods that were in the interface of the *incomplete* object, i.e., those in $\mathsf{M} \cup \mathsf{A}$. The record of methods in incgen is built by taking the concrete methods from the incomplete object o_1 and by taking the concrete version of the abstract methods from the complete object o_2. During this operation we must make sure that:

(*i*) methods from the complete object o_2 that are requested by the incomplete object o_1 get their *self* rebound to the new resulting composed object (this is the reason why we need to keep the generator also for complete object values).
(*ii*) methods of o_2 that are not requested by o_1 (we call these methods *additional*) are not subject to accidental overrides.

The second point, in particular, is crucial in our context, where *additional methods* in the complete object, "hidden" because of subsumption, may clash with methods already present in the incomplete object (i.e., those in M). The above two goals are achieved altogether using the generator component gen_1 inside incgen. This generator component builds a record where the *additional* methods (i.e., the ones belonging to $\mathsf{P} - \mathsf{A}$) are correctly bound, once and for all, to their implementation in the complete object (through s_2 that will be propagated with the auto-binding of self via fixpoint). The other methods (those requested by the incomplete object, i.e., belonging to $\mathsf{P} \cap \mathsf{A}$) rely on the rebound *self*, which, in turns, uses s_1 as a "handle" to hook onto the complete object method implementations. This gen_1 is therefore exploited to supply to v'_g (the generator of the complete object o_2) the "self" record, obtained by passing the new *self* to gen_1 and then applying the fixpoint. This realizes the main idea that the method bodies of the complete object will use as implementations of the *additional* methods the ones from the complete object and not possibly accidental homonyms from the incomplete object.

The rule (*completed*) transforms an incomplete object, for which all the missing methods are provided, or which is created by instantiating the class without abstract methods, into a corresponding complete one. Since at this stage the object is complete (i.e., it does not contain any abstract methods) we can apply the fixpoint operator to obtain the recursive record of methods invokable on that object. Notice that also in the complete object value the generator is still present since it can be used in further object compositions.

It might be tempting to argue that object composition is just syntactic sugar, i.e., it can be derived via an appropriate sequence of method additions, but this is not true. In fact, when adding a method, the method does not have a state, while a complete object used in an object composition has its own internal state (i.e., it has a private field, properly initialized when the object was created via "new" from the class). Being able to choose to complete an object via composition or via a sequence of method additions (of the same methods appearing in the complete object used in the composition) gives our calculus an extra bit of flexibility.

4.1 An example of reduction

Let us show how the object completion works through an example. Suppose we have the following objects (for simplicity we leave out the parameter of the methods, the private field, $\lambda y.\ldots y$, and dummy methods):

$$o_1 = \mathsf{obj}\langle v_g^1, \mathsf{M} = \{1\}, \mathsf{A} = \{2\}\rangle$$
$$o_2 = \mathsf{obj}\langle v_g^2, \mathsf{M} = \{1,2\}, \{m_1 = \lambda\, self.\ (self.m_1), m_2 = \lambda\, self.\ (self.m_1)\}\rangle$$
$$o = o_1 \longleftarrow o_2$$

where m_1 in o_2 is "hidden" (i.e., the type for o_2 will not contain the type of the method m_1 because of subsumption, see Section 5 for types). The object o_1 has m_2 as abstract, and it uses m_2 inside m_1 (with definition $m_1 = \lambda\, self.\ (self.m_2)$, visible in v_g^1 below). The program o loops infinitely, by first calling m_1 of o_1 (we recall that only methods belonging to the incomplete object interface are made accessible once completion has been performed). From rules (*new class*) and (*class*) we obtain the following generator for o_1 (we recall that m_2 is abstract in o_1):

$$v_g^1 = \lambda\, self.\ \left\{ \begin{array}{l} m_1 = (\lambda\, self.\ (self.m_2))\ self \\ m_2 = self.m_2 \end{array} \right\}$$

By applying rule (*obj comp*), o has the shape $\mathsf{obj}\langle \text{incgen}, \{1,2\}, \mathit{fix}(\text{incgen})\rangle$, where incgen is as follows:

$$\begin{array}{l} \mathsf{let}\ \text{incgen} = \\ \quad \mathsf{let}\ \text{gen}_1 = \lambda s_1.\lambda s_2.\left\{ \begin{array}{l} m_1 = (v_g^2\ s_2).m_1 \\ m_2 = s_1.m_2 \end{array} \right\} \quad \mathsf{in} \quad \lambda\, self.\left\{ \begin{array}{l} m_1 = (v_g^1\ self).m_1 \\ m_2 = (v_g^2\ \mathit{fix}(\text{gen}_1\ self)).m_2 \end{array} \right\} \end{array}$$

In the following we use the notation $o_i{::}m_j$ to refer to the (fully qualified) implementation of m_j in object (or incomplete object) o_i. If we invoke m_1 on o we want that $o_1{::}m_1$ is executed, then $o_2{::}m_2$, then $o_2{::}m_1$ (i.e., no accidental override take place), which will then loop on itself. We make explicit the reduction steps performed upon the invocation of the method m_1 on object o. (we denote $\text{gen}_1\mathit{fix}(\text{incgen})$ by gg):

$o.m_1 \to \mathit{fix}(\text{incgen}).m_1 \to (v_g^1\ \mathit{fix}(\text{incgen})).m_1 \to (\lambda\, self.\ (self.m_2))\mathit{fix}(\text{incgen}) \qquad o_1{::}m_1$: OK
$\to \mathit{fix}(\text{incgen}).m_2 \to (v_g^2\ \mathit{fix}(\text{gen}_1\ \mathit{fix}(\text{incgen}))).m_2 \to (v_g^2\ \mathit{fix}(gg)).m_2 \to$
$(\lambda\, self.\ (self.m_1))\mathit{fix}(gg) \qquad o_2{::}m_2$: OK
$\to \mathit{fix}(gg).m_1 \to (v_g^2\ \mathit{fix}(gg)).m_1 \to (\lambda\, self.\ (self.m_1))\mathit{fix}(gg) \qquad o_2{::}m_1$: OK
$\to \mathit{fix}(gg).m_1 \to (v_g^2\ \mathit{fix}(gg)).m_1 \to (\lambda\, self.\ (self.m_1))\mathit{fix}(gg) \qquad o_2{::}m_1$: OK
$\ldots$

We can see that each time the right implementation of the method was invoked and no accidental override took place, due to the usage of the additional generator gen_1.

5 Type system

Besides functional, record, and reference types, our type system has class types and object types (both for complete and incomplete objects):

$$\tau ::= \iota \mid \tau_1 \to \tau_2 \mid \tau\ \mathsf{ref} \mid \{m_i : \tau_{m_i}\}^{i\in I} \mid \mathsf{class}\langle \tau, \Sigma_\mathsf{M}, \Sigma_\mathsf{A}\rangle \mid \mathsf{obj}\langle \Sigma\rangle \mid \mathsf{obj}\langle \Sigma_\mathsf{M}, \Sigma_\mathsf{A}\rangle$$

$$\frac{typeof(const)=\tau}{\Gamma \vdash const:\tau}\ (const) \qquad \frac{}{\Gamma,x:\tau \vdash x:\tau}\ (proj) \qquad \frac{\Gamma,x:\tau \vdash e:\sigma}{\Gamma \vdash \lambda x.e:\tau\to\sigma}\ (\lambda)$$

$$\frac{\Gamma \vdash e_1:\tau\to\sigma \quad \Gamma \vdash e_2:\tau}{\Gamma \vdash e_1\ e_2:\sigma}\ (app) \qquad \frac{}{\Gamma \vdash fix:(\sigma\to\sigma)\to\sigma}\ (fix) \qquad \frac{\Gamma \vdash e:\tau \quad \Gamma \vdash \tau<:\sigma}{\Gamma \vdash e:\sigma}\ (sub)$$

$$\frac{\Gamma \vdash e_i:\tau_i}{\Gamma \vdash \{x_i=e_i\}^{i\in I}:\{x_i:\tau_i\}}\ (record) \qquad \frac{\Gamma \vdash e:\{x:\sigma\}}{\Gamma \vdash e.x:\sigma}\ (lookup)$$

$$\frac{}{\Gamma \vdash \mathsf{ref}:\tau\to\tau\ \mathsf{ref}}\ (ref) \qquad \frac{}{\Gamma \vdash\ !:\tau\ \mathsf{ref}\to\tau}\ (!) \qquad \frac{}{\Gamma \vdash :=\ :\tau\ \mathsf{ref}\to\tau\to\tau}\ (assign)$$

$$\frac{\Gamma' = \Gamma, x_1:\tau_1\ \mathsf{ref},\ldots,x_n:\tau_n\ \mathsf{ref} \quad \Gamma' \vdash v_i:\tau_i \quad \Gamma' \vdash e:\tau}{\Gamma \vdash H\langle x_1,v_1\rangle\ldots\langle x_n,v_n\rangle.e:\tau}\ (heap)$$

Table 7. Typing rules for expressions

where ι is a constant type, $\to$ is the functional type operator, τ ref is the type of locations containing a value of type τ. Σ (possibly with a subscript) denotes a record type of the form $\{m_i:\tau_{m_i}\}^{i\in I}$, $I\subseteq\mathbb{N}$. If $m_i:\tau_{m_i}\in\Sigma$ we say that the *label* m_i *occurs* in Σ (with type τ_{m_i}). $Labels(\Sigma)$ denotes the set of all the labels occurring in Σ.

Typing environments are defined as $\Gamma ::= \varepsilon \mid \Gamma,x:\tau \mid \Gamma,\iota_1<:\iota_2$ where $x\in Var$, τ is a well-formed type, ι_1,ι_2 are constant types, and $x,\iota_1\notin dom(\Gamma)$. *Typing judgments* are the following: $\Gamma\vdash\tau_1<:\tau_2$ (τ_1 is a subtype of τ_2), $\Gamma\vdash e:\tau$ (e has type τ).

Typing rules for lambda expressions, rules for expressions dealing with imperative side-effects via stores and rules for typing records are given in Table 7. We do not need any form of recursive types because we do not use a polymorphic *MyType* to type *self* (see, for instance, [7]). This prevents typing binary methods, but it still allows to type methods that modify *self*, which can be modelled as "void" methods.

Typing rules for class related forms are given in Table 8. In rule (*T class val*), $\mathsf{class}\langle\gamma,\{m_j:\tau_{m_j}\}^{j\in\mathsf{M}},\{m_i:\tau_{m_i}\}^{i\in\mathsf{A}}\rangle$ is the class type where γ is the type of the generator's argument. The two record types represent types of defined methods and abstract methods respectively. Thus, $\{m_i:\tau_{m_i}\}^{i\in\mathsf{M}\cup\mathsf{A}}$ is a record type representing the interface of the objects instantiated from the class. Rules (*T class val*) and (*T class*) assign the same type to their respective expressions, although deduced in a different way.

Table 9 shows the typing rules for manipulating objects. Incomplete objects are typed with the record type of defined methods and the record type of abstract methods (rule (*T inc obj*)). Notice that the type assigned to an incomplete object is similar to the one of the class the object is the instance of, but it does not contain information about the constructor. This is consistent with the fact that the constructor has already been called when an incomplete object has been created. We recall now the "dummy" methods introduced in Section 4, to justify their existence according to the typing rules: when typing an incomplete object value, "dummy" methods allow us to assign the type $\Sigma\to\Sigma$ to the generator v_g (the generator being a function from *self* to *self*). In fact, we recall that the body of "dummy" methods is a simple call to the homonym method on

$$\frac{\text{For } j \in \mathsf{M}\text{: } \Gamma \vdash v_{m_j} : \eta \to \{m_i : \tau_{m_i}\}^{i \in \mathsf{M} \cup \mathsf{A}} \to \tau_{m_j} \quad \Gamma \vdash v_c : \gamma \to \eta}{\begin{array}{ll} & \mathsf{class} \\ & \quad \mathsf{method}\ m_j = v_{m_j};\ ^{(j \in \mathsf{M})} \\ \Gamma \vdash & \quad \mathsf{abstract}\ m_i;\ ^{(i \in \mathsf{A})} \qquad : \mathsf{class}\langle \gamma, \{m_j : \tau_{m_j}\}^{j \in \mathsf{M}}, \{m_i : \tau_{m_i}\}^{i \in \mathsf{A}} \rangle \\ & \quad \mathsf{constructor}\ v_c \\ & \mathsf{end} \end{array}} \quad (T\ class)$$

$$\frac{\Gamma \vdash Gen_c : \gamma \to \{m_i : \tau_{m_i}\}^{i \in \mathsf{M} \cup \mathsf{A}} \to \{m_i : \tau_{m_i}\}^{i \in \mathsf{M} \cup \mathsf{A}}}{\Gamma \vdash \mathsf{classval}\langle Gen_c, \mathsf{M}, \mathsf{A} \rangle : \mathsf{class}\langle \gamma, \{m_j : \tau_{m_j}\}^{j \in \mathsf{M}}, \{m_i : \tau_{m_i}\}^{i \in \mathsf{A}} \rangle} \quad (T\ class\ val)$$

$$\frac{\Gamma \vdash e : \mathsf{class}\langle \gamma, \Sigma_\mathsf{M}, \Sigma_\mathsf{A} \rangle}{\Gamma \vdash \mathsf{new}\ e : \gamma \to \mathsf{obj}\langle \Sigma_\mathsf{M}, \Sigma_\mathsf{A} \rangle} \quad (T\ class\ inst)$$

Table 8. Typing rules for class related forms

$$\frac{\Gamma \vdash v_g : \{m_i : \tau_{m_i}\}^{i \in \mathsf{M} \cup \mathsf{A}} \to \{m_i : \tau_{m_i}\}^{i \in \mathsf{M} \cup \mathsf{A}}}{\Gamma \vdash \mathsf{obj}\langle v_g, \mathsf{M}, \mathsf{A} \rangle : \mathsf{obj}\langle \{m_j : \tau_{m_j}\}^{j \in \mathsf{M}}, \{m_i : \tau_{m_i}\}^{i \in \mathsf{A}} \rangle} \quad (T\ inc\ obj)$$

$$\frac{\begin{array}{l} \Gamma \vdash \{m_i = v_{m_i}\}^{i \in \mathsf{M}} : \{m_i : \tau_{m_i}\}^{i \in \mathsf{M}} \\ \Gamma \vdash v_g : \{m_i : \tau_{m_i}\}^{i \in \mathsf{M}} \to \{m_i : \tau_{m_i}\}^{i \in \mathsf{M}} \end{array}}{\Gamma \vdash \mathsf{obj}\langle v_g, \mathsf{M}, \{m_i = v_{m_i}\}^{i \in \mathsf{M}} \rangle : \mathsf{obj}\langle \{m_i : \tau_{m_i}\}^{i \in \mathsf{M}} \rangle} \quad (T\ obj) \qquad \frac{\Gamma \vdash e : \mathsf{obj}\langle \Sigma \rangle \qquad m_i : \tau_{m_i} \in \Sigma}{\Gamma \vdash e.m_i : \tau_{m_i}} \quad (T\ sel)$$

$$\frac{\Gamma \vdash e : \mathsf{obj}\langle \Sigma_\mathsf{M}, \Sigma_\mathsf{A} \rangle \quad m_l : \tau_{m_l} \in \Sigma_\mathsf{A} \quad \Gamma \vdash v_{m_l} : \Sigma_1 \to \tau_{m_l} \quad \Gamma \vdash (\Sigma_\mathsf{M} \cup \Sigma_\mathsf{A}) <: \Sigma_1}{\Gamma \vdash e \longleftarrow (m_l = v_{m_l}) : \mathsf{obj}\langle \Sigma_\mathsf{M} \cup \{m_l : \tau_{m_l}\}, \Sigma_\mathsf{A} - \{m_l : \tau_{m_l}\} \rangle} \quad (T\ meth\ add)$$

$$\frac{\Gamma \vdash e_1 : \mathsf{obj}\langle \Sigma_\mathsf{M}, \Sigma_\mathsf{A} \rangle \quad \Gamma \vdash e_2 : \mathsf{obj}\langle \Sigma_\mathsf{P} \rangle \quad \Gamma \vdash \Sigma_\mathsf{P} <: \Sigma_\mathsf{A} \quad Labels(\Sigma_\mathsf{P}) \cap Labels(\Sigma_\mathsf{M}) = \emptyset}{\Gamma \vdash e_1 \longleftarrow e_2 : \mathsf{obj}\langle \Sigma_\mathsf{M} \cup \Sigma_\mathsf{A} \rangle} \quad (T\ obj\ comp)$$

Table 9. Typing rules for object-related forms

self, so the type inferred for abstract methods is consistent with the types of "dummy" method bodies. Dummy methods appear only in the run-time semantics and are invisible to the programmer, thus they cannot be invoked.

Rule (*T obj*) says that the type of a complete object is the record of its method types. Notice that complete objects do not have a simple record type Σ, but an object type $\mathsf{obj}\langle\Sigma\rangle$. This is useful for distinguishing standard complete objects, which can be used for completing incomplete objects, from their internal auto-reference *self*, that has type Σ (in particular, this is to avoid *self-inflicted* object completions, unsound in our calculus). Note also that in the object expression, the first component v_g is a function from *self* to *self* (therefore typed with $\Sigma \to \Sigma$), because it works on the third component of the object, which is the record of object's methods. The only operation allowed on complete objects is method selection and it is typed as a record component selection (rule (*T sel*)).

$$\frac{}{\Gamma, \iota_1 <: \iota_2 \vdash \iota_1 <: \iota_2} \ (<: \textit{proj}) \qquad \frac{}{\Gamma \vdash \tau <: \tau} \ (<: \textit{refl})$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3} \ (<: \textit{trans}) \qquad \frac{\Gamma \vdash \tau' <: \tau \quad \Gamma \vdash \sigma <: \sigma'}{\Gamma \vdash \tau \to \sigma <: \tau' \to \sigma'} \ (<: \textit{arrow})$$

$$\frac{J \subseteq I}{\Gamma \vdash \{m_i : \sigma_i\}^{i \in I} <: \{m_j : \sigma_j\}^{j \in J}} \ (<: \textit{record}) \qquad \frac{\Gamma \vdash \Sigma <: \Sigma'}{\Gamma \vdash \mathsf{obj}\langle \Sigma \rangle <: \mathsf{obj}\langle \Sigma' \rangle} \ (<: \textit{cobj})$$

Table 10. Subtyping for objects

A method m_l can be added to an incomplete object (rule (*T meth add*)), only if this method is expected by the incomplete object (abstract method). Method addition in this case presents a sort of symmetry: the added method completes the functionalities of some already present methods, and may invoke some of them as well. Therefore, m_l's *self* type Σ_1 imposes some constraints on the type of the incomplete object that m_l is supposed to complete. Hence, the incomplete object must provide all the methods listed in Σ_1, on which the added method is parameterized. Σ_1 is inferred from m_l's body.

In the rule (*T obj comp*), Σ_P contains the type signatures of all the methods supported by the complete object (which may have more methods than those that are abstract in the incomplete object). The condition $\Sigma_\mathsf{P} <: \Sigma_\mathsf{A}$ ensures that the complete object contains at least all the methods needed to complete the incomplete object, and $\mathit{Labels}(\Sigma_\mathsf{P}) \cap \mathit{Labels}(\Sigma_\mathsf{M}) = \emptyset$ guarantees statically that there is no explicit name clash, i.e., there is no *true* conflict, as described in the Introduction. The type system does not rule out hidden conflicts introduced by subsumption, as these are not considered errors as long as they are taken care of dynamically, which is the goal of our run-time semantics. The resulting complete object contains the signatures of all the methods of the incomplete object.

The subtyping relation for record and object types is given in Table 10. For uniformity with respect to object types, we define only width subtyping on record types as well. However, modifying the subtyping rules in order to allow depth subtyping on record types only would be just a technicality and an orthogonal issue with respect to the subject of the paper, therefore we leave this modification out for the sake of clarity.

6 A more flexible solution

In the solution presented so far, the interface of an object resulting from an object composition is dictated by the incomplete object only, in the sense that, in the resulting composed object, only the methods of the incomplete object are invocable by an external user. Such a restriction on object composition was not present in the previous versions of the incomplete objects [1, 2] and it is not necessary to solve the problems of dynamic name clashes, although it actually simplifies its treatment, and it allows to implement an "early-binding" version of the "privacy-via-subsumption" notion of [11].

In this section, we present an alternative solution that removes this restriction and is thus more flexible (in the sense that the interface of the composed object will contain

$$\frac{\Gamma \vdash e_1 : \mathsf{obj}\langle \Sigma_\mathsf{M}, \Sigma_\mathsf{A} \rangle \quad \Gamma \vdash e_2 : \mathsf{obj}\langle \Sigma_\mathsf{P} \rangle \quad \Gamma \vdash \Sigma_\mathsf{P} <: \Sigma_\mathsf{A} \quad \mathit{Labels}(\Sigma_\mathsf{P}) \cap \mathit{Labels}(\Sigma_\mathsf{M}) = \emptyset}{\Gamma \vdash e_1 \longleftarrow\!\!+\ e_2 : \mathsf{obj}\langle \Sigma_\mathsf{M} \cup \Sigma_\mathsf{P} \rangle} \quad (\textit{T obj comp})$$

$$\begin{array}{l} \mathsf{obj}\langle v_g, \mathsf{M}, \mathsf{A} \rangle \longleftarrow\!\!+\ \mathsf{obj}\langle v'_g, \mathsf{P}, \{m_i = v_{m_i}\}^{i \in \mathsf{P}} \rangle \rightarrow \\ \mathsf{let\ incgen} = \\ \quad \mathsf{let\ gen}_1 = \lambda s_1 . \lambda s_2 . \left\{ \begin{array}{ll} m_l = \lambda y.\ (v'_g\ s_2).m_l\ y & (l \in \mathsf{P} \cap \mathsf{M}) \\ m_r = \lambda y.\ s_1.m_r\ y & (r \in \mathsf{P} - \mathsf{M}) \end{array} \right\} \ \mathsf{in} \\ \qquad \lambda \mathit{self} . \left\{ \begin{array}{ll} m_j = \lambda y.\ (v_g\ \mathit{self}).m_j\ y & (j \in \mathsf{M}) \\ m_i = \lambda y.\ (v'_g\ \mathit{fix}(\mathrm{gen}_1\ \mathit{self})).m_i\ y & (i \in \mathsf{P} - \mathsf{M}) \end{array} \right\} \mathsf{in} \\ \qquad \mathsf{obj}\langle \mathrm{incgen}, \mathsf{M} \cup \mathsf{A}, \mathit{fix}(\mathrm{incgen}) \rangle \end{array} \quad (\textit{obj comp})$$

Table 11. The typing and reduction rule to change

more methods). From the point of view of typing, all we have to do is to change the typing rule for object composition in order to include all these object methods, see rule (*T obj comp*) in Table 11 (which mirrors the original rule of [2]). Now, the semantic rule for object composition must be changed, since we do not hide all the *additional* methods: those that do not clash with methods defined by the incomplete object must be visible in the resulting object (while those that clash are obviously hidden). All we have to do is to modify slightly the rule (*obj comp*), obtaining the one given in Table 11. Notice that all of the above is enough to obtain a fully-fledged second solution.

7 Conclusions

In this paper we presented two possible solutions to solve the "method composition versus width subtyping" conflict. We remark that the high-level ideas underpinning our solutions are general. In particular, the idea of having self basically "split" into two parts when composing two objects, one taking care of the statically bound methods, the other one dealing with the dynamically bound ones, can be applied within any setting presenting the same problem. We would also like to stress the flexibility of our approach, i.e., giving the incomplete object-calculus an ML-like based operational semantics: working directly with class-as-generator-functions and objects-as-records (and their respective types) allows us to alternate between one version of the calculus and another with minimal changes, both in the typing rules and in the semantics. In particular, since the operational semantics is a set of rewriting rules into functions, we can manipulate functions to achieve our goals, instead of using ad-hoc changes to the semantics. Moreover, with respect to the dictionaries of [11] in the late-binding setting (where the host object is substituted for self in order to solve the self autoreferences at method-invocation time), our early-binding setting (where the host object is bound to self at object-creation time) allows a corresponding solution that is more oriented to an implementation and, in particular, would not suffer from overheads due to dictionary management and lookups, as the original calculus does, as pointed out in [11] itself. This is true also for the alternative solution.

The approach we chose here was to allow width subtyping on complete objects only. It is possible to have width subtyping on incomplete objects as well, if hidden

method names are carried along: (*i*) in the type of the object; (*ii*) in the object itself. Solution (*i*) would imply a more restrictive typing rule for object composition, to also check the possible conflicts among non-hidden and hidden methods, and rule out such conflicts completely. We think, though, that such a solution is too restrictive, as we think this kind of name clash is not an error. Hidden method name information in the object (solution (*ii*)) would solve all possible ambiguities at run-time, but it would be less standard, as the subsumption rule would act on the object expression, not only on its type. Nevertheless, we think this solution has the advantage of being quite general, even though it might be considered not elegant, and it will be presented as future work.

As a future work plan, we are also considering the integration of a form of object-based override with a form of depth subtyping on object types, and we will study solutions to deal with the conflicts arising. In particular, an incomplete object could redefine a method that is provided through method addition or object composition. This operation is the concept of method redefinition/overriding of class-based inheritance adapted to an object-based setting. This kind of method override will enhance dynamic compositionality and flexibility and will allow the programmer to implement rather easily a chain of method invocation established at run-time (see, e.g., *decorator* and *chain of responsibilities* patterns [9]). Furthermore, we will study a composition operation between two complete objects (e.g., no abstract methods).

Acknowledgements. The authors would like to thank the anonymous referees.

References

1. L. Bettini, V. Bono, and S. Likavec. A core calculus of mixin-based incomplete objects. In *Proc. FOOL 11*, pages 29–41, 2004.
2. L. Bettini, V. Bono, and S. Likavec. Safe and Flexible Objects. In *Proc. SAC '05, OOPS track*, pages 1258–1263. ACM Press, 2005.
3. V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In *Proc. ECOOP '99*, pages 43–66. LNCS 1628, Springer-Verlag, 1999.
4. V. Bono, A. Patel, V. Shmatikov, and J. C. Mitchell. A core calculus of classes and objects. In *Proc. MFPS '99*, volume 220, 1999.
5. E. Crank and M. Felleisen. Parameter-passing and the lambda calculus. In *Proc. POPL '91*, pages 233–244, 1991.
6. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
7. K. Fisher, F. Honsell, and J. C. Mitchell. A lambda-calculus of objects and method specialization. *Nordic J. of Computing*, 1(1):3–37, 1994.
8. K. Fisher and J. C. Mitchell. A delegation-based object calculus with subtyping. In *Proc. 10th International Conference on Fundamentals of Computation Theory (FCT '95)*, pages 42–61. LNCS 965, Springer-Verlag, 1995.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
10. I. Mason and C. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proc. ICALP '89*, pages 574–588. LNCS 372, Springer-Verlag, 1989.
11. J. G. Riecke and C. A. Stone. Privacy via subsumption. *Information and Computation*, 172(1):2–28, 2002. A preliminary version appeared in FOOL5.
12. A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.